Apuntes U 4

Tema ***Algoritmos de Ordenamiento y Número Aleatorio***

"Estar preparado es importante, saber esperar lo es aún más, pero aprovechar el momento adecuado es la clave de la vida."
Arthur Schnitzler

¿Qué es el ordenamiento?

Para conseguir mayor eficiencia en el tratamiento de la información, tanto en el ámbito de almacenamiento, la información debe tener algún tipo de ordenamiento.

> En computación *y matemáticas* un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, ***el resultado de salida ha de ser una permutación,*** *(o reordenamiento)* de la entrada que satisfaga la relación de orden dada.

Criterios:
- Numéricos: Ascendente/Descendente
- Lexicográficos: Ascendente/Descendente *(refiriéndonos a caracteres)*

Métodos de ordenamiento según su lugar de procesamiento:

Internos: Se denomina así porque se realiza directamente y completamente en memoria principal *(todos los objetos que se ordenan caben en la memoria principal de la computadora)* con lo que el proceso es más rápido.

Externos: Cuando no cabe toda la información en memoria principal y es necesario ocupar memoria secundaria *(disco)*. El ordenamiento ocurre transfiriendo bloques de información a memoria principal en donde se ordena el bloque y este es regresado, ya ordenado, a memoria secundaria. En consecuencia es mas lento.

Métodos de ordenamiento según su forma de procesamiento:

Iterativos: Estos métodos son simples de entender y de programar ya que son iterativos, simples ciclos y sentencias que hacen que el vector pueda ser ordenado.
Recursivos: Estos métodos son aún más complejos, requieren de mayor atención y conocimiento para ser entendidos. Son rápidos y efectivos, utilizan generalmente la técnica Divide y vencerás, que consiste en

dividir un problema grande en varios pequeños para que sea más fácil resolverlos. Mediante llamadas recursivas a si mismos, es posible que el tiempo de ejecución y de ordenación sea más óptimo.

Complejidad en los Métodos de ordenamiento

La complejidad de un algoritmo es la cantidad de trabajo realizado y se mide por el número de operaciones básicas que se han llevado a cabo.

Por lo cual podemos comenzar comparando algunos de los algoritmos que desarrollaremos en esta unidad:

INSERCIÓN	cuadrático
SHELLSORT	sub- cuadrático
QUICKSORT	logarítmico
MERGESORT	logarítmico

Ordenamiento por Burbuja (Bubble Sort) $O(n^2)$

La Ordenación de burbuja (Bubble Sort) es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista o arreglo que va a ser ordenado, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.

Algoritmo BURBUJA_1

```
Inicio
        desde (I=1 hasta (N-1)) hacer
                desde (J=1 hasta (N-1)) hacer
                        Si (X(j) > X(j+1)) entonces
                                AUX=X(j)
                                X(j) =X(j+1)
                                X(j+1)=AUX
                        fin_si
                fin_desde
        fin_desde
fin
```

Algoritmo BURBUJA_2

```
inicio
        desde (I=1 hasta (N-1)) hacer
                desde (J=1 hasta (N-I)) hacer
                        Si (X(j) > X(j+1)) entonces
                                AUX=X(j)
                                X(j) =X(j+1)
                                X(j+1)=AUX
                        fin_si
                fin_desde
        fin_desde
fin
```

Algoritmo BURBUJA_3

```
inicio
        MARCA=FALSO
        I=1
        mientras ((MARCA=FALSO) y (I< N-1)) hacer
                MARCA=VERDADERO
                desde (J=1 hasta (N-1)) hacer
                        Si (X(j) > X(j+1)) entonces
                                AUX=X(j)
                                X(j)=X(j+1)
                                X(j+1)=AUX
                                MARCA=FALSO
                        fin_si
                fin_desde
                I=I+1
        fin_mientras
fin
```

Ordenamiento por insersión (Insertion Sort) $\boldsymbol{O(n^2)\Omega(n)}$

En este método, también llamado **insertion sort**, cuando hay n elementos se supone que se tiene un segmento inicial del array ordenado, el paso general es aumentar la longitud del segmento ordenado insertando el elemento siguiente X en el lugar adecuado, esto se hace moviendo cada elemento del segmento ordenado a la derecha, hasta que se encuentra un elemento < X.

```
Algoritmo INSERCION
        desde (N hasta I=2) hacer
                X=L(I)
```

```
                J=I- 1
                Mientras ( (L(J) > X) and (J > 0)) hacer
                        L(J+1)=L(J)
                        J=J-1
                fin_mientras
                L(J+1)=X
        fin_desde
fin
```

Nota: Partimos del supuesto de que ya se agrando el Arreglo y el número insertado ya se encuentra en la última posición.

```
Algoritmo INSERCION-2
        inserto=0; I=N-1;
        mientras  (I>1) o (inserto=0) hacer
                si X>=L(I) entonces
                                L(I+1)=X;
                                inserto=1;
                        sino
                                L(I+1)=L(I);                    finsi
                I--;
                si (I=1) y (inserto=0) entonces
                        L(I)=X;
                finsi
        fin_mientras
Fin
```

Para hallar la complejidad del algoritmo, analizamos el peor, el mejor y el caso medio.
El peor caso, es decir, el mayor número de comparaciones, se daría cuando el bucle interno tiene que hacer el mayor número de veces (I-1) siendo I el valor del núcleo externo, es decir, cuando cada elemento examinado es menor que el resto y esto ocurrirá cuando el array esté ordenado en orden contrario al que deseamos.
En el caso medio las comparaciones que se harían serían: para cada elemento I examinando como mínimo realizo una comparación en el caso que sea mayor que el resto y como máximo N-I.

Ordenamiento Shell (Shell Sort) $O(n^2)$

El método se denomina Shell en honor de su inventor Donald Shell. Su implementación original, requiere $O(n^2)$ comparaciones e intercambios en el peor caso. Un cambio menor presentado en el libro de V. Pratt produce una implementación con un rendimiento de $O(n \log 2\, n)$ en el peor caso. Esto es mejor que las $O(n^2)$ comparaciones requeridas por algoritmos simples pero peor que el óptimo $O(n\ log\ n)$. Aunque es fácil desarrollar un sentido intuitivo de cómo funciona este algoritmo, es muy difícil analizar su tiempo de ejecución.

El Shell Sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

-El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".

-El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

El algoritmo **Shell Sort** mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del **Shell Sort** es un simple *ordenamiento por inserción*, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

La **secuencia de espacios** es una parte integral del ***Shell sort***. Cualquier secuencia incremental funcionaría siempre que el último elemento sea 1. El algoritmo comienza realizando un ordenamiento por inserción con espacio, siendo el espacio el primer número en la secuencia de espacios. Continúa para realizar un ordenamiento por inserción con espacio para cada número en la secuencia, hasta que termina con un espacio de 1. Cuando el espacio es 1, el ordenamiento por inserción con espacio es simplemente un ordenamiento por inserción ordinario, garantizando que la lista final estará ordenada.

Pasos a seguir

*Dividir la lista original en **n/2** grupos de dos, considerando un incremento o salto entre los elementos de **n/2**.

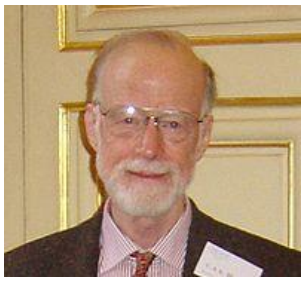
*Analizar cada grupo por separado, comparando las parejas de los elementos, y si no están ordenados, se intercambian.

*Se divide ahora la lista en la mitad de grupos **(n/4)**, con un incremento o salto entre los elementos también mitad **(n/4)**, y nuevamente se clasifica cada grupo por separado.

Así sucesivamente se sigue dividiendo la lista en la mitad de grupos que el recorrido anterior con un incremento de salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado

El algoritmo termina cuando se consigue el tamaño de salto 1.

Ordenamiento Rápido (Quick Sort) $O(n \log n)$



El ordenamiento rápido *(Quicksort)*, o también ordenamiento por Partición, es un algoritmo creado por el científico británico en computación C. A. R. Hoare basado en la técnica de divide y vencerás, que permite, en promedio, ordenar **n** elementos en un tiempo proporcional a **n log n**.

El algoritmo: Se elige un elemento de la lista de elementos a ordenar, al que llamaremos **pivote**.
Se reacomodan los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el **pivote** *ocupa exactamente el lugar que le corresponderá en la lista ordenada*.
La lista queda **separada en dos sublistas**, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

La eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

- ***En el mejor caso***, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es O(n·log n).
- ***En el peor caso***, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente.
- ***En el caso promedio***, el orden es O(n·log n).

Técnicas de elección del pivote: El algoritmo básico del método Quicksort consiste en tomar cualquier elemento de la lista al cual denominaremos como pivote, dependiendo de la partición en que se elija, el algoritmo será más o menos eficiente.

- Tomar un elemento cualquiera como pivote tiene la ventaja de no requerir ningún cálculo adicional, lo cual lo hace bastante rápido. Sin embargo, esta elección *"a ciegas"* siempre provoca que el algoritmo tenga un orden de $O(n^2)$ para ciertas permutaciones de los elementos en la lista.
- Otra opción puede ser recorrer la lista para saber de antemano qué elemento ocupará la posición central de la lista, para elegirlo como pivote. Esto puede hacerse en **O(n)** y asegura que hasta en el peor de los casos, el algoritmo sea **O(n·log n)**. No obstante, el cálculo adicional rebaja bastante la eficiencia del algoritmo en el caso promedio.
- La opción a medio camino es tomar tres elementos de la lista - por ejemplo, el primero, el segundo, y el último - y compararlos, eligiendo el valor del medio como pivote.

```
Procedimiento QUICKSORT(Li,Ls:entero;L:array[1..N])
var PIVOTE:entero
inicio
        si (Li < Ls) entonces
                SUBLISTAS (Li,Ls,PIVOTE,L:array)
                QUICKSORT (Li,PIVOTE-1,L:array)
                QUICKSORT (PIVOTE+1,Ls,L:array)
                fin_si
        fin_quicksort
```

```
Procedimiento SUBLISTAS(FIRST,LAST,PIV:entero,L:array)
var X,I:entero
inicio
        X ☐ L(FIRST)
        PIV ☐FIRST
        desde (I=FIRST+1 hasta LAST)
                si (L(I) < X) entonces
                        (PIV ☐ PIV+1)
                        AUX ☐ L(PIV)
                        L(PIV)☐ L(I)
                        L(I) ☐ AUX
                fin_si
        fin_desde
        L(FIRST)☐ L(PIV)
        L(PIV) ☐ X
fin
```

```
Algoritmo PRIN_QUICKSORT
var a:array [1..N]
inicio
        leer A
        QUICKSORT(1,N,A
)
        escribir A
fin
```

Algoritmo	**Operaciones máximas**
Burbuja	$O(N^2)$
Inserción	$O(N^2)/4$
Selección	$O(N^2)$
Shellsort	$O(N\ log^2 N)$
Mergesort	$O(N \log N)$
Quicksort	$O(N^2)$, en el peor y $N \log N$ en el promedio de casos

Ordenamiento Selección rápida

Una característica especial es la SELECCIÓN, la cual tiene una complejidad tal vez menor que el ordenamiento. Esto es debido principalmente a que debemos "encontrar" un elemento dentro de un arreglo y no tenemos la complejidad del ordenar al resto.
Lo ideal es poder utilizar el QUICKSORT como base y realizarle los ajustes necesarios.
Como vimos anteriormente, el QUICKSORT hace uso de dos llamadas recursivas, por lo cual en este caso sólo necesitaríamos una. Además podemos entender que en el peor de los casos de la SELECCIÓN RAPIDA estaríamos con las características del QUICKSORT.
Nuevamente, la elección del pivote, es el factor de éxito de este proceso.

Otros algoritmos

Los conceptos se utilizan para valorar las páginas, se dividen en dos grandes familias:

1.- Criterios internos
2.- Criterios externos

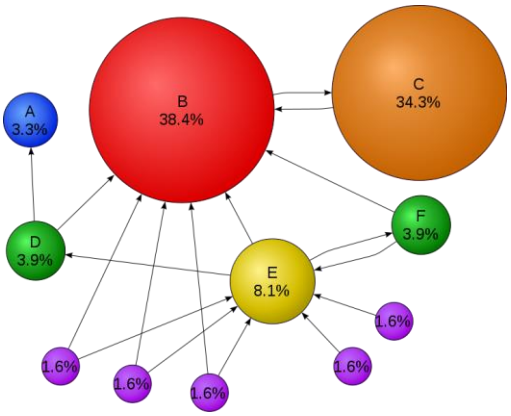
Tanto los conceptos internos a la propia página web como los externos son importantes, algunos de los conceptos incluso no se valoran directamente, si no que permiten que la página consiga una mejor valoración para otras.
Hay que saber qué es lo que se pretende con una página antes de definir qué estrategia o combinación de estrategias se aplicará.

Criterios internos a la página web:
A grandes rasgos, consisten en todo aquello que se encuentra en la propia página, entre ellos destacan:
1. Número de pantallas: (además pantallas más valoración)
2. Títulos de las pantallas: (las pantallas deben tener un título)
3. Redacción de los textos
4. Estructura que permita una fácil navegación
5. Tecnologías aplicadas: (evitar utilización abusiva de flash, javascripts...)
6. Direcciones de las pantallas de acuerdo con el contenido: (url descriptivas).

Criterios externos a la página web: Son los conceptos que los buscadores interpretan según cuanto más enlaces tiene una página, más relevancia merece, y por lo tanto aparecerá más arriba en las búsquedas relacionadas.
Estos enlaces pueden realizarse de varias maneras, aun cuando algunos no tienen valor para los buscadores, la explicación en este caso se limitará a aquellas que si que tienen. De un lado, existe la posibilidad de enlaces mediante imágenes (.GIF o bien .JPG); estas imágenes, si no contienen textos alternativos ni descripción, simplemente votan a la página de destino, incrementando el valor de cara a los motores de búsqueda.
El otro tipo de enlace, es el enlace de texto, que al cliquear encima, lleva hacia la página web de destino. Este tipo de enlace es interesante porque potencia la página de destino por el texto exacto del enlace, permitiendo potenciar exclusivamente aquel texto, además de también incrementar el valor de la página de destino por su propio contenido.

Por este motivo, si hay muchas páginas que enlazan una página, los buscadores interpretan que cada página web emite su voto para aquella a la que va destinado el enlace, dándole más y más preferencia. De este modo, las páginas destinatarias de enlaces, ganan posiciones por ser consideradas como más importantes por aquellos términos por los que están referidas, aún así, es necesario saber cómo deben ser dichos enlaces para optimizar su utilidad.

PageRank es una marca registrada y patentada por *Google el 9 de enero de 1999* que ampara una familia de algoritmos utilizados para asignar de forma numérica la relevancia de los documentos (o páginas web) indexados por un motor de búsqueda. Sus propiedades son muy discutidas por los expertos en optimización de motores de búsqueda. El sistema PageRank es utilizado por el popular motor de búsqueda Google para ayudarle a determinar la importancia o relevancia de una página. Fue desarrollado por los fundadores de Google, Larry Page *(apellido, del cual, recibe el nombre este algoritmo)* y Sergey Brin, en la Universidad de Stanford mientras estudiaban el posgrado en ciencias de la computación.



PageRank confía en la naturaleza democrática de la web utilizando su vasta estructura de enlaces como un indicador del valor de una página en concreto. Google interpreta un enlace de una página A a una página B como un voto, de la página A, para la página B. Pero Google mira más allá del volumen de votos, o enlaces que una página recibe; también analiza la página que emite el voto. Los votos emitidos por las páginas consideradas "importantes", es decir con un PageRank elevado, valen más, y ayudan a hacer a otras páginas "importantes". Por lo tanto, el PageRank de una página refleja la importancia de la misma en Internet.

$$\mathrm{PR}(A) = (1-d) + d\sum_{i=1}^{n}\frac{\mathrm{PR}(i)}{C(i)}$$

Donde:

- $\mathrm{PR}(A)$ es el PageRank de la página A.
- **d** es un factor de amortiguación que tiene un valor entre 0 y 1.
- $\mathrm{PR}(i)$ son los valores de PageRank que tienen cada una de las páginas **i** que enlazan a A.
- **C(i)** es el número total de enlaces salientes de la página i (sean o no hacia A).

Número Aleatorio

Un número aleatorio es un resultado de una variable al azar especificada por una función de distribución. Cuando no se especifica ninguna distribución, se presupone que se utiliza la distribución uniforme continua en el intervalo [0,1]

Los números aleatorios permiten a los modelos matemáticos representar la realidad.
En general cuando se requiere una **impredecibilidad** en unos determinados datos, se utilizan números aleatorios
Los seres humanos vivimos en un medio aleatorio y nuestro comportamiento lo es también. Si deseamos predecir el comportamiento de un material, de un fenómeno climatológico o de un grupo humano podemos inferir a partir de datos estadísticos. Para lograr una mejor aproximación a la realidad nuestra herramienta predictiva debe funcionar de manera similar: **aleatoriamente**. De esa necesidad surgieron los modelos de simulación.
En la vida cotidiana se utilizan números aleatorios en situaciones tan dispares como pueden ser los juegos de azar, en el diseño de la caída de los copos de nieve, en una animación por ordenador, en tests para localización de errores en chips, en la transmisión de datos desde un satélite o en las finanzas.

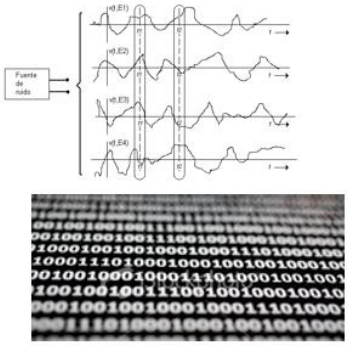
Como se pueden generar los números aleatorios

Para generarlos podemos aprovecharnos de situaciones reales para obtener una tabla de números aleatorios, como la lista de los números de Lotería Nacional premiados a lo largo de su historia, pues se caracterizan por que cada dígito tiene la misma probabilidad de ser elegido, y su elección es independiente de las demás extracciones.

Se piensa que las personas son generadores aleatorios imperfectos, hay estudios que demuestran que existen tendencias claras en los humanos para la elaboración de ***secuencias sesgadas*** y están relacionadas con características personales, con los conocimientos o informaciones previas o con la edad.

Métodos manuales: *lanzamiento de monedas, lanzamientos de dados, dispositivos mecánicos, dispositivos electrónicos.*
Métodos de computación analógica: *son métodos que dependen de ciertos procesos físicos aleatorios, por ejemplo, el comportamiento de una corriente eléctrica.*
Tablas de bibliotecas: *son números aleatorios que se han publicado; de los cuales podemos encontrar listas en los libros de probabilidad y tablas de matemáticas. Estos números fueron generados por alguno de los métodos de computación analógica.*

Métodos de computación digital: *cuando se usa una computadora digital.*



Números psedudoaleatorios

Son unos números generados por medio de una ***función*** *(determinista, no aleatoria)* y que *aparentan* ser aleatorios. Estos números pseudoaleatorios se generan a partir de un **valor inicial** aplicando iterativamente la función. La sucesión de números pseudoaleatorios es sometida a diversos tests para medir hasta qué punto se asemeja a una sucesión aleatoria .

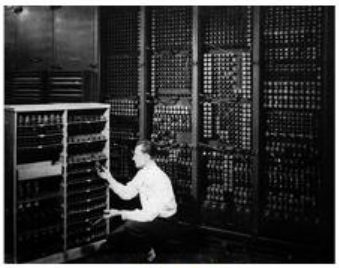
¿Por qué hay que recurrir a los número pseudoaleatorios?

Fundamentalmente porque las sucesiones de números pseudoaleatorios son más rápidas de generar que las de números aleatorios. Si las personas tenemos dificultad en generar números aleatorios, mucho más la tiene una computadora, la dificultad está en que una computadora es tan "torpe" que no sabe generarlos *(es solo una metáfora)*. Por eso usan números pseudoaleatorios, que para nuestro fin es lo mismo, pues nadie los puede predecir.

Método del cuadrado medio

Ante las enormes posibilidades científicas que se vislumbraban ante el creciente uso de las primeras grandes computadoras electromecánicas *(MARK I, ENIAC, UNIVAC, etc.)*, las cuales mediante el uso de tarjetas perforadas o cintas magnéticas permitían ejecutar diversas instrucciones digitales para realizar operaciones muy complejas, entró en escena el matemático ***John von Neumann (1903–1957)***, quien hacia 1946 propuso usar el algoritmo conocido como el **"Método del Cuadrado Medio"** para la generación matemática de números pseudoaleatorios.


MARK I


ENIAC


UNIVAC

El Método del Cuadrado Medio de von Neumann consistía en el uso de un algoritmo en el cual al inicio se introduce un número cualquiera conformado por 10 dígitos, luego se calcula el cuadrado de ese número inicial, a continuación se toman exactamente los 10 dígitos ubicados en la mitad del número resultante, y ese número conformado por los 10 dígitos se toma como un nuevo número aleatorio que sirve para engrosar la secuencia aleatoria generada, al cual posteriormente se le puede aplicar de nuevo el algoritmo del cuadrado medio para así obtener sucesivamente más números aleatorios.

El esquema de funcionamiento del algoritmo del Método del Cuadrado Medio propuesto por John von Neumann es el siguiente:

MÉTODO DEL CUADRADO MEDIO PARA GENERAR NÚMEROS ALEATORIOS:			
No. de arranque:	Resultado al ser elevado al cuadrado (x^2):	Selección de los 10 dígitos del medio:	Nuevo número aleatorio generado:
5772156649	33317792380594909201	33317792380594909201	7923805949
7923805949	62786700717407800000	62786700717407800000	7007174078
7007174078	49100488559395200000	49100488559395200000	4885593952
4885593952	23869028263819000000	23869028263819000000	0282638190

En esta tabla se observa que cada nuevo número aleatorio que al final es generado por la aplicación del algoritmo del Cuadrado Medio puede luego ser tomado como un nuevo número de arranque (o "Número Semilla") para volver a generar un nuevo número aleatorio, proceso que se supone es perpetuo hasta el infinito y con resultados siempre impredecibles.

John von Neumann tenía la gran esperanza de que con la ayuda de las nuevas computadoras que se estaban construyendo se podrían diseñar y usar algoritmos cada vez más complejos, que servirían para producir listados de números aleatorios que al ser sometidos a los Tests Estadísticos más comunes no reflejarían la presencia de la más mínima *tendencia*, *desviación*, *regresión* o *repetición periódica*.
Sin embargo, el mismo ***Método del Cuadrado Medio***. más tarde reflejó que en cierto momento la secuencia de los números aleatorios generados puede caer en un **"loop"**, es decir, autónomamente cae en un ciclo repetitivo de resultados que se denomina **"periodo"**, lo cual le resta el pretendido carácter aleatorio e impredecible a los números generados por ese método.

*Esto ocurre principalmente cuando en cierto momento el número **cero (0)** comienza a aparecer muchas veces dentro de los **10 dígitos** que conforman el **supuesto número aleatorio** resultante de la elevación al*

cuadrado, caso en el cual cuando esos ceros son elevados de nuevo al cuadrado seguirán repitiéndose en mayor medida y por siempre dentro de la secuencia de los nuevos números generados.
*Esto lo demostró **G. E. Forsythe** en los años 50's al aplicar el Método del Cuadrado Medio a números que dentro de los 10 dígitos incluían bastantes ceros, con lo que obtuvo secuencias repetitivas de números generados que al poco tiempo **se hacían fácilmente predecibles**.*

Otros matemáticos probaron el **Método del Cuadrado Medio** usando números de hasta 20 dígitos, o incluso usando números binarios, y se logró establecer que **existen 13 posibles ciclos o periodos repetitivos** diferentes en los que pueden caer las secuencias de los números generados por este método, y además **se probó que cada uno de estos periodos puede comenzar a repetirse de manera variable**, es decir, bien puede comenzar a repetirse cada 150 resultados aparecidos, o bien puede comenzar a repetirse cada 2.000 resultados generados, o bien cada 12.345 resultados, etc., todo lo cual depende de los ***"Números–Semilla"*** que inicialmente sean introducidos como arranque para dar inicio a este algoritmo.
En otras palabras, si los números generados por el Método del Cuadrado Medio terminan reflejando una secuencia repetitiva o un ciclo que ocurre periódicamente cada tantos resultados aparecidos *(un "loop")*, entonces esos números no deberían ser considerados completamente aleatorios sino ***sólo pseudoaleatorios***.

$$R_{n+1} = \text{mid}(R_n^2, m)$$

where:
R_{n+1} = *new random number*
R_n = *previous random number*
R_0 = *initial seed value*
mid(,m) = *m number of digits extracted from the relative middle*

Generadores de números aleatorios

Un paso clave en simulación es tener algoritmos que generen variables aleatorias con distribuciones específicas, como la exponencial, normal, etc.
Esto es efectuado en dos fases:

- ***La primera consiste en generar una secuencia de números aleatorios distribuidos "uniformemente" entre 0 y 1.***
- ***Luego esta secuencia es transformada para obtener los valores aleatorios de las distribuciones deseadas.***

Las propiedades deseadas del generador son las siguientes:

1. **Deben ser eficientes computacionalmente**: dado que típicamente se requieren varios miles de números aleatorios por corrida, el tiempo de procesador requerido para generarlos debe ser pequeño.

2. **El periodo debe ser largo**: periodos cortos limitan la longitud aprovechable de una corrida de simulación porque el reciclaje resulta en una repetición de secuencias de eventos.

3. <u>Los valores sucesivos deben ser independientes y uniformemente distribuidos</u>: la correlación entre números sucesivos debe ser pequeña y si es significante indica dependencia.

Generadores Congruenciales Lineales (GCL)

Los principales generadores de números pseudoaleatorios utilizados en la actualidad son los llamados generadores ***congruenciales lineales***, introducidos por **Lehmer en 1951**. Un método congruencial comienza con un valor inicial *(semilla)* x_0, y los sucesivos valores x_n, para $n \geq 1$, se obtienen con la siguiente fórmula:

$$x_n = ax_{n-1} + b \; mod \; m$$

Donde $\boldsymbol{a}, \boldsymbol{m}, \; \boldsymbol{b}$ son enteros positivos que se denominan, respectivamente, el **multiplicador**, el **módulo** y el **incremento**. Si $b = 0$ el generador se denomina multiplicativo; en caso contrario se denomina mixto.
La selección de $a, m, \; b$ afectan el periodo y la autocorrelación en la secuencia.

Si m es primo, distintas elecciones de a, permiten generar un periodo completo de (m-1) valores.

Entre los resultados de los estudios realizados con estos generadores tenemos:
1. El módulo m debe ser grande. Dado que los x están entre 0 y m-1, el periodo nunca puede ser mayor que m.
2. Para que el computo de mod m sea eficiente, m debe ser una potencia de 2, es decir, $\boldsymbol{2^k}$. En este caso mod m puede ser obtenido truncando el resultado y tomando en k bits a la derecha.
3. Si b es diferente de cero, el periodo máximo posible m se obtiene si y sólo si:

a) Los enteros m y b son primos relativos -- no tengan factores comunes excepto el 1.
b) Todo número primo que sea un factor de m lo es también de a-1.
c) a-1 es un múltiplo de 4 si m es un múltiplo de 4.

Todas estas condiciones se cumplen si $m = 2^k$, a = 4c + 1, y b es impar, donde c, b, y k son enteros positivos.
Si un generador tiene el periodo máximo posible se llama generador de **periodo completo**.
Todos los generadores de periodo completo no son igualmente buenos. Son preferibles los generadores con **menor autocorrelación** entre números sucesivos.
Por ejemplo, los dos generadores siguientes son de periodo completo, pero el primero tiene una correlación de 0.25 entre x_{n-1} y x_n , mientras que el segundo tiene una correlación despreciable de menos de 2^{-18}.

$$x_n = \left((2^{34} + 1)x_{n-1} + 1\right) mod \; 2^{35}$$
$$x_n = \left((2^{18} + 1)x_{n-1} + 1\right) mod \; 2^{35}$$

```
programa prueba_random;
        variables
                i,x: entero;
```

```
        ran: real;
funcion random(var x:entero): real;
        constantes
                a = 16807; # multiplicador
                m = 2147483647; # modulo
                q = 127773; {#m div a
                r = 2836; # m mod a
        comienzo
                x = a*(x mod q) - r*(x div q);
                si (x < 0) hacer x = x + m;
                random = x/m;
        fin;
comienzo
        x:=1; #semilla
        para i=1 a 10000 hacer ran=random(x);
        escribir(x);
        #Salida = 1043618065
fin.
```

Generadores de Fibonacci extendidos.

Una secuencia de Fibonacci $\{x_n\}$ se genera por la siguiente relación:

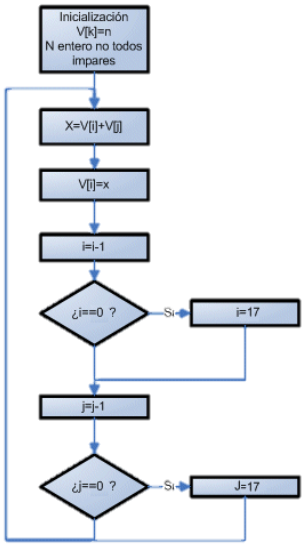
$$x_n = x_{n-1} + x_{n-2}$$

Se puede intentar usar un generador de la forma:

$$\boldsymbol{x_n = (x_{n-1} + x_{n-2}) mod\ n}$$

Sin embargo esta secuencia no tiene buenas propiedades aleatorias y en particular tiene ***alta correlación serial***. El siguiente generador, que sigue este enfoque, pasa la mayoría de las pruebas estadísticas:

$$\boldsymbol{x_n = (x_{n-5} + x_{n-17}) mod\ 2^k}$$



Para implementar este generador se pueden usar 17 localidades de memoria L[1], ..., L[17] las cuales son inicializadas con 17 enteros que no sean todos pares.
Fijamos i y j en 5 y 17 respectivamente, y el siguiente procedimiento es ejecutado para obtener los números aleatorios:

x= L[i] + L[j];
L[i]= x;
i= i - 1; si i=0 entonces i= 17;
j= j - 1; si j=0 entonces j= 17;

La adición en la primera línea es automáticamente $\boldsymbol{mod\ 2^k}$ en máquinas de k-bits y Aritmética de complemento a 2.
El periodo del generador es $\boldsymbol{2^k\left(2^{17}-1\right)}$ que es considerablemente mayor al que se puede obtener con GCL.

Generadores Combinados.

Es posible combinar generadores para obtener "mejores" generadores. Algunas de las técnicas usadas son:

1. **OR-exclusivo** de números aleatorios de dos o más generadores. Esta técnica es similar a la anterior excepto que la suma es reemplazada por un *or-exclusivo* bit por bit. Se ha demostrado que esta técnica aplicada a números ligeramente aleatorios puede ser usada para generar números con mayor aleatoriedad.
2. **Barajeo**. Usa una secuencia como un índice para decidir qué número generado por otra secuencia será retornado. Por ejemplo, uno de estos algoritmos usa un arreglo de tamaño 100 que contiene números de una secuencia aleatoria xn. Para generar un número aleatorio se genera un número aleatorio yn (entre 0 y m-1) para obtener el índice i = 1 + 99yn / (m-1). El valor del i-esimo elemento del arreglo es devuelto. Un nuevo valor xn es calculado y almacenado en la i-esima localidad.

Aunque estos generadores caen dentro de la sección anterior, su relevancia amerita su discusión por separado.

Consideremos el generador $x_n = 7^5 x_{n-1}\ mod(2^{31} - 1)$

En la medida que las computadoras se han vuelto más rápidas, la longitud de su ciclo se ha tornado inadecuada ya que se corre el riesgo de que, en unas cuantas horas de simulación, la secuencia se agote y se repita varias veces, trayendo como consecuencia serias dudas en cuanto a la validez de los resultados.

Supongamos que tenemos una máquina con un procesador con 109 ciclos o tics del reloj por segundo. Supongamos también que el procesador es capaz de generar un número aleatorio por *ciclo (esto es muy optimista ya que en realidad se requieren varios ciclos para producir un número aleatorio debido a las operaciones involucradas)*.

Bajo estas condiciones se agotaría la secuencia en $2{,}1 \cdot \frac{10^9}{10^9} = 2.1$ seg

Por supuestos que será en más tiempo, pero se observa que efectivamente esta secuencia es fácilmente agotable durante una simulación.

Una manera para conseguir generadores con periodos más largos consiste en sumar números aleatorios de dos o más generadores.

Generadores Normalmente Usados.

Un generador GCL muy popular y que ya hemos mencionado es:

$$x_n = 7^5 x_{n-1}\ mod(2^{31} - 1)$$

Éste es usado en el sistema SIMPL/I de IBM (1972), APL de IBM (1971), el sistema operativo PRIMOS de Prime Computer (1984), y la librería científica de IMSL (1987). Tiene buenas propiedades aleatorias y es recomendado como un estándar mínimo.

Un estudio de generadores multiplicativos GCL con módulo $m = 2^{31} - 1$ que comparó su eficiencia y aleatoriedad, recomienda los dos siguientes como los mejores:

$$x_n = 48271 x_{n-1}\ mod(2^{31} - 1)$$

$$x_n = 69621x_{n-1}\ mod(2^{31} - 1)$$

El siguiente generador es usado en SIMSCRIPT II.5 y en FORTRAN DEC-20:

$$x_n = 6630360016x_{n-1}\ mod(2^{31} - 1)$$

El siguiente generador es usado en el sistema Pascal de la Universidad de Sheffield para computadores Prime:

$$x_n = 16807x_{n-1}\ mod\ 2^{31}$$

Dado que 16.807 no es de la forma 8i ± 3, este generador no tiene el periodo máximo posible de 2^{31-2}. También es usado en la subrutina UNIFORM del paquete estadístico SAS, pero una técnica de barajeo es usada para mejorar la aleatoriedad.

SIMULA en UNIVAC usa:

$$x_n = 5^{13}x_{n-1}2^{35}$$

Algunos autores dicen que no tiene buenas propiedades aleatorias.

El sistema operativo UNIX soporta en siguiente GCL mixto:

$$x_n = (1103515245x_{n-1} + 12345)mod\ 2^{32}$$

***NOTA**: Diseñar nuevos generadores parece muy simple, pero muchos generadores propuestos por expertos estadísticos fueron encontrados deficientes. Por lo tanto es mejor usar un generador que ha sido extensamente probado en vez de inventar uno nuevo.*

Selección de Semilla.

En principio la semilla no debería afectar los resultados de la simulación. Sin embargo, una mala combinación de semilla y generador pueden producir conclusiones erróneas.
Si el generador es de periodo completo y solo se requiere una variable aleatoria, cualquier semilla es buena. Hay que tener especial cuidado en simulaciones que requieren números aleatorios para más de una variable (simulaciones de secuencias múltiples), que es la mayoría de los casos. Por ejemplo, la simulación de una cola simple requiere generar llegadas y servicios aleatorios y requiere dos secuencias de números aleatorios.
1. **No use cero**. Cero funciona para generadores GCL mixtos pero hace que los multiplicativos se queden en cero.
2. **Evite valores pares**. Si un generador no es de periodo completo (por ejemplo GCL multiplicativo con modulo $\mathrm{m} = 2^k$) la semilla debe ser impar. En otros casos no importa.
3. **No subdivida una secuencia**. Usar una única secuencia para todas las variables es un error común. Por ejemplo, en la secuencia {u1 , u2 , ...} generada a partir de la semilla u0 , el analista usa u1 para generar el tiempo entre llegadas, u2 para el tiempo de servicio, etc. Esto puede resultar en una fuerte correlación entre las variables.

4. **Use secuencias que no se solapan**. Cada secuencia requiere su semilla. Si la semilla es tal que hace que dos secuencias se solapen, habrá correlación entre las secuencias y estas no serán independientes.
Consideremos el ejemplo trivial de iniciar las dos secuencias de una cola simple con la misma semilla, lo cual haría las secuencias idénticas e introduciría una fuerte correlación positiva entre los tiempos entre llegadas y los tiempos de servicio. Esto puede llevar a conclusiones erróneas.
Hay que seleccionar las semillas de forma tal que las secuencias no se solapen. Si {u1 , u2 , ...} generada a partir de la semilla u0 , y necesitamos por ejemplo 10.000 tiempos entre llegadas, 10.000 tiempos de servicios, etc., podemos seleccionar u0 como la semilla de la primera secuencia, u10.000 para la segunda, u20.000 para la tercera, etc. Los un se pueden determinar mediante un programa de prueba que llama al generador o se pueden calcular directamente para generadores GCL mixtos o multiplicativos (b = 0) con la formula siempre que los cálculos sean exactos:

$$x_n = a^n x_0 + \frac{b(a^n - 1)}{a - 1} mod\ m$$

5. **Reuse semillas en replicaciones sucesivas**. Si el experimento es replicado varias veces, la secuencia no necesita ser reinicializada y se puede usar la semilla dejada en la replicación previa.
6. **No use semillas aleatorias**. Semillas aleatorias, como por ejemplo la hora del día, causan dos problemas:
   - La simulación no puede ser reproducida.
   - No se puede garantizar que secuencias múltiples no se solapen.

Números aleatorios con distribución no uniforme.

Una variable aleatoria es una función que asume sus valores de acuerdo a los resultados de un experimento aleatorio, es decir, un experimento donde existe incertidumbre acerca del resultado que va a ocurrir.

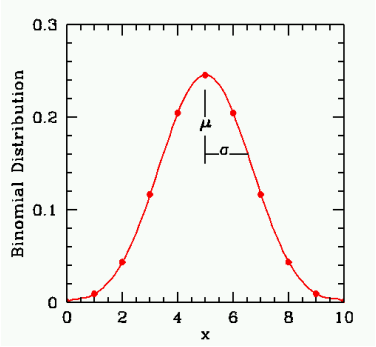
Una ***variable aleatoria es discreta*** si su rango de valores es un conjunto finito o infinito enumerable.
Existen una infinidad de variables aleatorias discretas, entre las más conocidas están: ***la Binomial, la geométrica, la hipergeométrica, la Poisson y la Binomial Negativa***.
Si la variable aleatoria discreta X tiene rango de valores RX entonces la función **p(k)=Prob[X=k]** donde **x∈RX** es llamada la función de probabilidad de **X**.
Asimismo, la función $F(t) = P[X \leq t] = \sum_{k \leq t} P[X = k]$ es llamada la función de distribución acumulativa de X.
Una ***variable aleatoria continua*** es aquella cuyo rango de valores es cualquier intervalo de la recta real, entre las más conocidas están: ***la uniforme, la exponencial, la gamma, la Ji-Cuadrado, la Beta, la Normal, la t de Student, la Cauchy, la Weibull,etc***.
Si la variable aleatoria continua X tiene rango de valores RX entonces existe una función no-negativa $f(x)$ tal que $P(a < X < b) = \int_a^b f(x)dx$
Asimismo, la función

$$F(t) = P[X \leq t] = \int_{-\infty}^{t} f(x)dx$$

es llamada la función de distribución acumulativa de X.
En los modelos estocásticos existirán una o más variable aleatorias interactuando.

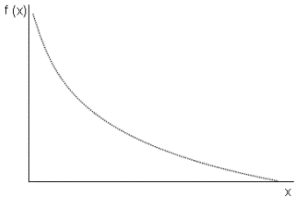
Estas variables siguen distribuciones de probabilidad teóricas o empíricas, diferentes a la distribución uniforme (0-1).

Binomial: Es una distribución de probabilidad discreta que cuenta el número de éxitos en una secuencia de n ensayos de Bernoulli independientes entre sí, con una probabilidad fija p de ocurrencia del éxito entre los ensayos. Un experimento de Bernoulli se caracteriza por ser dicotómico, esto es, sólo son posibles dos resultados. A uno de estos se denomina éxito y tiene una probabilidad de ocurrencia p y al otro, fracaso, con una probabilidad q = 1 - p. En la distribución binomial el anterior experimento se repite n veces, de forma independiente, y se trata de calcular la probabilidad de un determinado número de éxitos. Para n = 1, la binomial se convierte, de hecho, en una distribución de Bernoulli.

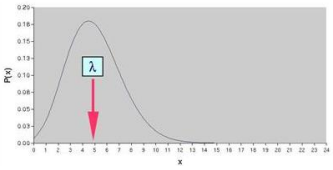


Geométrica: Es cualquiera de las dos distribuciones de probabilidad discretas siguientes:
- la distribución de probabilidad del número X del ensayo de Bernoulli necesaria para obtener un éxito, contenido en el conjunto { 1, 2, 3,...} o
- la distribución de probabilidad del número Y = X − 1 de fallos antes del primer éxito, contenido en el conjunto { 0, 1, 2, 3,... }.

Cual de éstas es la que uno llama "la" distribución geométrica, es una cuestión de convención y conveniencia.



Poisson: es una distribución de probabilidad discreta que expresa, a partir de una frecuencia de ocurrencia media, la probabilidad de que ocurra un determinado número de eventos durante cierto período de tiempo. Concretamente, se especializa en la probabilidad de ocurrencia de sucesos con probabilidades muy pequeñas, o sucesos "raros".



Para generar números que sigan el comportamiento de estas variables, se pueden utilizar algunos métodos como los siguientes:

1. Método de la transformada inversa
2. Método de rechazo
3. Método de composición
4. Procedimientos especiales

Método de la transformada inversa: El método de la transformada inversa utiliza la distribución acumulada F(x) de la distribución que se va a simular.
Puesto que F(x) está definida en el intervalo (0-1), se puede generar un número aleatorio uniforme R y tratar de determinar el valor de la variable aleatoria para cual su distribución acumulada es igual a R, es decir, el valor simulado de la variable aleatoria que sigue un distribución de probabilidad f(x), se determina al resolver la siguiente ecuación.

$$F(x) = R \text{ ó } x = F^{\wedge -1}(R)$$

La dificultad principal de este método descansa en el hecho de que en algunas ocasiones es difícil encontrar la transformada inversa. Sin embargo, si esta función inversa ya ha sido establecida, generando números aleatorios uniformes se podrán obtener valores de la variable aleatoria que sigan la distribución de probabilidad deseada.

Método de aceptación y rechazo: La idea aquí es que se tiene una variable aleatoria Y con función de densidad g(y), la cual puede ser generada fácilmente.
Se desea generar otra variable X con función de densidad f(x), para ello se genera un valor de Y con densidad g(y) y se toma x=Y con probabilidad proporcional a f(y)/g(y).
Esto es: $\mathrm{P}[X = y] = k\, f(y)/g(y)$ donde k es la constante de proporcionalidad.
Si c es una constante tal que $f(y)/g(y) \leq c$ para todo y
Entonces el siguiente sería el algoritmo de aceptación y rechazo para generar una variable aleatoria X con función de densidad f(x).

Paso 1: Generar Y con densidad g
Paso 2: Generar una variable aleatoria uniforme U(0,1)
Paso 3: Si $\mathrm{U} \leq f(y)/cg(y)$ entonces X=Y de lo contrario volver al paso 1

Mientras más cerca se encuentre f de g más rápidamente se obtendrá la cantidad deseada de valores aleatorios de X.
Usualmente se toma la densidad uniforme como la densidad g(y) y en ese caso el método de aceptación y rechazo es llamado “hit and miss”.

Método de composición-descomposición: Se basa en la idea de dividir la f(x) original en una combinación de fi (x) cuya selección se hace en
base a minimizar el tiempo de computación requerido.
Algoritmo:

* Dividir la f(x) original en sub-áreas.
* Definir una distribución de probabilidad para cada sub-área:
* Expresar la distribución original como:
* Obtener la distribución acumulada de las áreas
* Generar dos números aleatorios uniformemente distribuidos, R1 y R2.
*Con R1 entrar a la distribución acumulada de las áreas (por el eje y) y seleccionar cuál fi (x) se va a usar (método de la transformada inversa).

· Utilizar R2 para simular x por el método de la transformada inversa con fi (x).
· Repetir generando nuevos pares de números aleatorios.

Generación de permutaciones aleatorias.

Pensando en un juego de azar, por ejemplo un juego de cartas; uno de los jugadores tiene el objetivo de mezclar el mazo, el resto de los jugadores “verifican” que no pueda realizar ninguna acción deshonesta. Pero, ¿qué pasa con la llegada de Internet y los juegos que se realizan en la red, por ejemplo los juegos de póker?
En este tipo de juego, los jugadores no pueden realizar esta acción de “verificar” de la misma manera que en un juego presencial.

Pues bien, esa "verificación" puede ser garantizada por dos caminos:

• Por quien cubre el rol de casino on-line y todos los jugadores aceptan esa posición y entienden que es una posición de privilegio.

• Por un protocolo de computación criptográfico multiparte, el cual asegura que ningún jugador tendrá alguna ventaja con el resto, considerando que cada participante "conocerá" sólo una parte del "secreto" que equivale al reparto de cartas.

Dos consideraciones debemos hacer en este esquema:

a- Existe un grupo de participantes **P** = P1,……,Pn

b- Existe un secreto **s**.

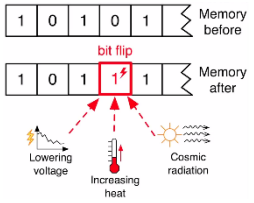
Algoritmos aleatorios.

La NASA ha dicho que hay un hardware generador de números aleatorios en el Rover. Luego de eso, hicieron otro comentario, recordándonos que solo necesitamos que esos algoritmos **funcionen en la práctica**. Para que algo funcione en práctica, significa que hay siempre alguna **posibilidad de error**, pero tal vez la **probabilidad es tan pequeña** que la ignoramos en la practica, y si eso suena loco, solo tenemos que darnos cuenta de que en el mundo físico nada es incuestionable, siempre hay posibilidad de error.

Por ejemplo, el empaquetado de chips contiene pequeñas cantidades de contaminantes radioactivos, y cuando estos decaen, se liberan partículas alfa las cuales pueden en realidad cambiar bits en memoria, y quizás cambiar un número inesperadamente. Aún mas interesante, los rayos cósmicos pueden también causar pequeños errores, nunca podemos borrar la posibilidad de error completamente. ¿Pero cual es exactamente una probabilidad de error **aceptable** para la NASA?.

La NASA dice, "**Solo tenemos que estar seguros que, la probabilidad de error para una prueba dada, sea menor que la probabilidad de ganar la lotería dos veces seguidas**".

$$e = 6 \times 10^{-14}$$



Suficientemente seguro; no esperaremos ver un error, y esto podría correr cientos o miles de veces.

Ahora podemos preguntarnos, ¿el acceso a la aleatoriedad nos ayudaría a acelerar un algoritmo de decisión tal como esta prueba preliminar?

Por ejemplo, el empaquetado de chips contiene pequeñas cantidades de contaminantes radioactivos, y cuando estos decaen, se liberan partículas alfa las cuales pueden en realidad cambiar bits en memoria, y quizás cambiar un número inesperadamente. Aún mas interesante, los rayos cósmicos pueden también causar pequeños errores, nunca podemos borrar la posibilidad de error completamente. ¿Pero cual es exactamente una probabilidad de error **aceptable** para la NASA?.

Un algoritmo aleatorio (o probabilístico, o no determinista) es un algoritmo que basa su resultado en la toma de algunas decisiones al azar, de tal forma que, en promedio, obtiene una buena solución al problema planteado para cualquier distribución de los datos de entrada. Es decir, al contrario que un algoritmo determinista, a partir de unos mismos datos se pueden obtener distintas soluciones y, en algunos casos, soluciones erróneas.

- Uso de probabilidades en el análisis del algoritmo
- Usa generadores de números aleatorios
- La "aleatoriedad" es la característica clave
- Usado para problemas NP-Hard o NP-Completos

Ejemplo de algoritmo aleatorio:

```
findA(array A, n)
begin
    repeat
        randomly select one element out of n elements.
    until 'X' is found.
end
```

Existen varios tipos de algoritmos probabilísticos dependiendo de su funcionamiento, pudiéndose distinguir:

Algoritmos numéricos, que proporcionan una solución aproximada del problema.

Algoritmos de Montecarlo, que pueden dar la respuesta correcta o respuesta erróneas (con probabilidad baja). El algoritmo Montecarlo falla con cierta probabilidad, pero no puede decir cuando falla. Podemos reducir la probabilidad de falla significativamente, por medio de la ejecución repetitiva pudiendo tomar la mayoría de las respuestas produce un si o un no.

Algoritmos de Las Vegas, que nunca dan una respuesta incorrecta: o bien no encuentran la respuesta correcta e informan del fallo. El algoritmo de Las vegas falla con cierta probabilidad, pero nos avisa cuando falla. Podemos correrlo una y otra vez hasta que tenga éxito. En otras palabras, Las vegas es un algoritmo que corre por un tiempo indeterminado y no predecible pero siempre tiene éxito.

Algoritmos aleatorios (Montecarlo).

Bajo el nombre de ***"Método de Monte Carlo"*** se agrupan una serie de procedimientos que analizan distribuciones de variables aleatorias usando simulación de números aleatorios. El Método de Monte Carlo da solución a una gran variedad de problemas matemáticos haciendo experimentos con muestreos estadísticos en una comput adora. El método es aplicable a cualquier tipo de problema, ya sea ***estocástico o determinístico***.

Generalmente en Estadística los modelos aleatorios se usan para simular fenómenos que poseen algún componente aleatorio, pero en el método de Monte Carlo, el objeto de la investigación es el objeto en sí mismo, un suceso aleatorio o pseudo-aleatorio se usa para estudiar el modelo.
A veces la aplicación del método de Monte Carlo se usa para analizar problemas que no tienen un componente aleatorio explícito; en estos casos un parámetro determinista del problema se expresa como una distribución aleatoria y se simula dicha distribución.
La simulación de Monte Carlo fue creada para resolver integrales que no se pueden resolver por métodos analíticos, para resolver estas integrales se usaron números aleatorios.
El nombre y el desarrollo sistemático de los métodos de Monte Carlo data aproximadamente de 1944 con el desarrollo de la computadora electrónica. Sin embargo, hay varias instancias (aisladas y no desarrolladas) en muchas ocasiones anteriores a 1944.
El uso real de los métodos de Monte Carlo como una herramienta de investigación, viene del trabajo de la bomba atómica durante la Segunda Guerra Mundial.

Variantes del Método de Monte Carlo se usan para resolver problemas muy diversos. De todas ellas, en el caso de los cálculos computacionales relacionados con sistemas moleculares, las más importantes son las siguientes:
1-Método Clásico (Classical Monte Carlo, CMC): aplicación de distribuciones de probabilidades (generalmente la distribución clásica de Maxwell y Boltzmann) para obtener propiedades termodinámicas, estructuras de energía mínima y constantes cinéticas.
2-Método Cuántico (Quantum Monte Carlo, QMC): uso de trayectorias aleatorias para calcular funciones de onda y energías de sistemas cuánticos y para calcular estructuras electrónicas usando como punto de partida la ecuación de Schroedinger.
3-Método de la Integral a lo largo de la Trayectoria (Path-Integral Quantum Monte Carlo, PIMC): cálculo de las integrales de la Mecánica Estadística Cuántica para obtener propiedades termodinámicas y constantes cinéticas usando como punto de partida la integral a lo largo de la trayectoria de Feynman.
4-Método Volumétrico (Volumetric Monte Carlo, VMC): uso de números aleatorios y cuasi-aleatorios para generar volúmenes moleculares y muestras del espacio de fase molecular).
5-Método de Simulación (Simulation Monte Carlo, SMC): uso de algoritmos aleatorios para generar las condiciones iniciales de la simulación de trayectorias cuasi-clásicas o para introducir efectos estocásticos ("termalización de las trayectorias") en Dinámica Molecular. (El así llamado "Método Cinético" —Kinetic Monte Carlo, KMC— es uno de los SMC.).

Algoritmos aleatorios (Las Vegas).

Un algoritmo tipo Las Vegas es un algoritmo de computación de carácter aleatorio (random) que no es aproximado: es decir, da el resultado correcto o informa que ha fallado.
Un algoritmo de este tipo no especula con el resultado sino que especula con los recursos a utilizar en su computación.

De la misma manera que el método de Montecarlo, la probabilidad de encontrar una solución correcta aumenta con el tiempo empleado en obtenerla y el número de muestreos utilizado. Un algoritmo tipo Las Vegas se utiliza sobre todo en problemas NP-completos, que serían intratables con métodos determinísticos.
Existe un riesgo de no encontrar solución debido a que se hacen elecciones de rutas aleatorias que pueden no llevar a ningún sitio. El objetivo es minimizar la probabilidad de no encontrar la solución, tomando decisiones aleatorias con inteligencia, pero minimizando también el tiempo de ejecución al aplicarse sobre el espacio de información aleatoria.
La clase de complejidad de los problemas de decisión de estos algoritmos con ejecución polinómica es ZPP.
ZPP=RP*no-RP

Su esquema de implementación se parece al de los algoritmos de Montecarlo, pero se diferencian de ellos en que incluyen una variable booleana para saber si se ha encontrado la solución correcta.

Algoritmos aleatorios (Las Vegas vs Montecarlo).

?	?	?	?	?	?	A	?	?	?

Arreglo para verificar si tiene un valor determinado

```
Function LasVegasSearch(A array; n char; size:integer);
Begin
        result=false;
        While (result<>true) do
        Begin
                pos=random(size);
                if (n=A[pos]) then
                        result=true;
        end
        return result;
end
```

```
Function MonteCarloSearch(A array; n char; size, x:integer);
Begin
        i=0;
        result=false;
        While (i <= x) do
        Begin
                i=i+1;
                pos=random(size);
                if (n=A[pos]) then
                        result=true;
        end
        return result;
end
```

Una mejora que se le puede hacer es cortar cuando lo encuentre

Test Aleatorio de Primalidad

Un **test de primalidad** (o chequeo de primalidad) es un algoritmo que, dado un número de entrada n, no consigue verificar la hipótesis de un teorema cuya conclusión es que n es compuesto. Esto es, un test de primalidad sólo conjetura que ***"ante la falta de certificación sobre la hipótesis de que n es compuesto podemos tener cierta confianza en que se trata de un número primo".***

Esta definición supone un grado menor de confianza que lo que se denomina **prueba de primalidad** *(o test verdadero de primalidad)*, que ofrece una seguridad matemática al respecto.

Ahora bien, ***¿cuál es nuestra necesidad o importancia de corroborar si un número es o no un número primo?***
Su importancia radica, por ejemplo, en que actualmente los métodos criptográficos usan números primos de muchas cifras decimales (*llamados primos industriales*) como parte fundamental del proceso de encriptación.
El mayor problema es que la seguridad del método se diluye cuando elegimos un número que creemos es primo cuando sin embargo no lo es, por lo que es fundamental tener algoritmos rápidos y eficientes que certifiquen la primalidad.

Sea n ∈ N. n es primo si y sólo si los únicos divisores que posee son 1 y n

Esta definición se generaliza para los dominios de integridad, lo cual da lugar al concepto de elemento primo. Otra definición podría ser:
Sea D un dominio de integridad y sea n ∈ D. n es un elemento primo si y sólo si:

a) n = 0
b) n no es una unidad.
c) Si p divide a ab con a, b ∈ D entonces p\|a o bien p\|b.

Ahora que tenemos unas definiciones debemos poder certificar si un número es primo o no.
Uno de los primeros algoritmos para probar la primalidad es el que indica que:
Como input toma n ∈ Z

Paso. 1 Escribir todos los números hasta n.
Paso. 2 Tachar el 1 pues es una unidad.
Paso. 3 Repetir hasta n

Tachar los múltiplos del siguiente número sin tachar, excepto el mismo número.
Output = Los números tachados son compuestos y los no tachados son primos.

Este algoritmo tiene la particularidad de que no sólo nos certifica si un número es primo o no, sino que nos da todos los primos menores que él.
El algoritmo funciona pues estamos comprobando si es o no múltiplo de algún número menor que el, lo cual es equivalente a la definición de número primo.
Este método es óptimo cuando necesitamos construir la tabla con todos los números primos hasta n, pero ***es tremendamente ineficiente para nuestro propósito****, pues se trata de un algoritmo exponencial tanto en tiempo como en espacio.*

<u>División por tentativa - Trial division *(o test de divisibilidad)*</u>: Consiste en estudiar la divisibilidad de un número impar y se toma el hecho de que ningún numero mayor a 3 es primo si no es divisible entre ningún otro impar menor o igual que $\sqrt{N}$.
Es un algoritmo sencillo para comprobar la primalidad de un número. Es muy rápido, pero no sirve para números grandes*(64 bits)*, solo para números pequeños *(32 bits)*. Ya que se necesita comprobar cerca de $\sqrt{N}/_2$ divisores, empleándose una cantidad de tiempo del orden de $O(\sqrt{N})$.
La división por tentativa garantiza encontrar un factor de *n*, puesto que comprueba todos los factores primos posibles de *n*. Por tanto, si el algoritmo no encuentra ningún factor, es una prueba de que *n* es primo.

<u>Pequeño teorema de Fermat</u>: El pequeño teorema de Fermat da una condición necesaria para que un número p sea primo.

Si p es un número primo y $0 < A < P$ entonces, $A^{P-1} \equiv 1(mod\ P)$.
Si consideramos un k cualquiera verificado $1 < k < P$. Como P es primo y menor que A y k, $Ak \equiv 0(mod\ P)$ es imposible. Conisderemos cualquier $1 \leq i \leq j < P$. $Ai \equiv Aj(mod\ P)$ implicaria $A(j-i) \equiv 0(mod\ P)$ pero esto es imposible dado el argumento anterior, ya que $1 \leq j - i < P$.

Si el recíproco del Teorema pequeño de Fermat fuese cierto, entonces tendríamos un test de primalidad computacionalmente equivalente a la exponenciación modular (esto es, $O(\log N)$). Desafortunadamente ***el resultado recíproco no es cierto***.

Otro algoritmo elemental es el de divisiones sucesivas o de **Fibonacci**:
Input = n ∈ Z
Paso. 1 Mientras i < √n
Si n = 0 mod i entonces Output = Compuesto
Paso. 2 Output = Primo
Este algoritmo es exponencial en tiempo
Ambos algoritmos fueron de los primeros utilizados para comprobar la primalidad.

Avanzando en el tiempo podemos mencionar el Test de **Miller-Rabin**:
Input = n, k ∈ Z /* k = número máximo de repeticiones */
Paso. 1 Escribir n − 1 = 2sd dividiendo sucesivas veces n − 1 entre 2.
Paso. 2 Desde i = 1 hasta k hacer
Sea a un número escogido aleatoriamente en {1, . . . , n − 1}
Si ad = 1 mod n entonces Output = Compuesto.
Desde r = 0 hasta s − 1 hacer
• Si a2rd = −1 mod n entonces Output = Compuesto.
Paso. 3 Output = Probable Primo.
Este algoritmo es del orden O(log2 (n))

Ahora veremos el algoritmo conocido como **AKS**, el cual distingue entre números primos y compuestos
Input = n ∈ Z

Paso. 1 Si n = ab para algún a ∈ N y b > 1 entonces Output = Compuesto.
Paso. 2 Encuentra el menor r tal que or(n) > 4 log2(n)
Paso. 3 Si 1 < (a, n) < n para algún a =< r entonces Output = Compuesto.
Paso. 4 Si n =< r entonces Output = Primo.
Paso. 5 Desde a = 1 hasta 2√p(r) log(n) comprueba
si (x + a)n = xn + a mod (xr − 1, n) entonces Output = Compuesto.
Paso. 6 Output = Primo

Los autores demostraron además que, si determinados números primos *(llamados números primos de Sophie Germain)* tienen la distribución conjeturada por el matemático austriaco Emil Artin, el exponente 21/2 que aparece en la expresión de complejidad puede reducirse a 15/2. Lo que implica que el tiempo estimado de ejecución sería equivalente a O((log n)6+e) de forma incondicional.
En realidad este descubrimiento no tiene implicaciones prácticas en la computación moderna. Lo cierto es que las partes constantes de la complejidad del algoritmo son mucho más costosas que en los actuales algoritmos probabilísticos. Es de esperar que en el futuro cercano se obtengan mejoras en esas constantes, pero lo cierto es que los algoritmos actuales de generación de números primos cubren bastante bien las necesidades actuales y, posiblemente, las futuras (y es poco probable que la línea propuesta mejore en tiempo de ejecución a los algoritmos probabilísticos existentes). Sin embargo sí que tiene una importancia fundamental desde el punto de vista teórico, ya que supone la primera prueba de primalidad de estas características que ha sido matemáticamente demostrada.